CSCI 103
More Recursion, Linked List Recursion, and Generating All Combinations

Mark Redekopp

Tracing Recursive Algorithms

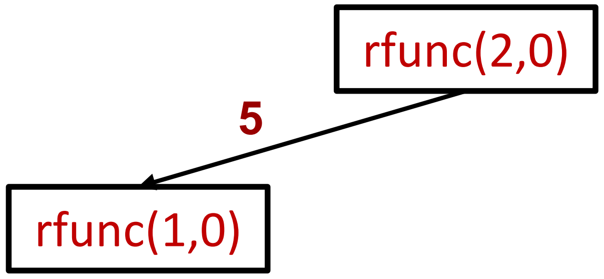
Tracing Recommendations

- Show the call tree
  – Draw each instance of a recursive function as a box and list the inputs passed to it
  – When you hit a recursive call draw a new box with an arrow to it and label the arrow with the line number of where you left off in the caller

Analyze These!

```
00: void rfunc(int n, int t) {
01:    if (n == 0) {
02:      cout << t << " ";
03:      return;
04:   }
05:   rfunc(n-1, 3*t);
06:   rfunc(n-1, 3*t+2);
07:   rfunc(n-1, 3*t+1);
08: }
09: int main() {
10:    rfunc(2, 0);
11: }
```

- What does this function print? Show the call tree?



- What is the runtime in terms of n?

Analyze These!

- What does this function return for g(3122013)

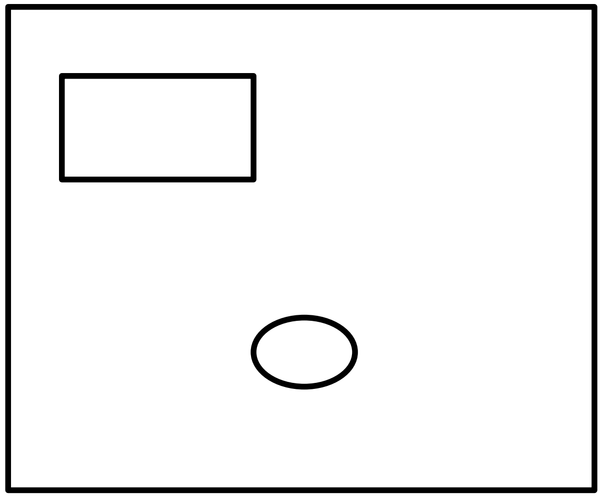
```
int g(int n)
{
   if (n % 2 == 0)
      return n/10;
   return g(g(n/10));
}
```

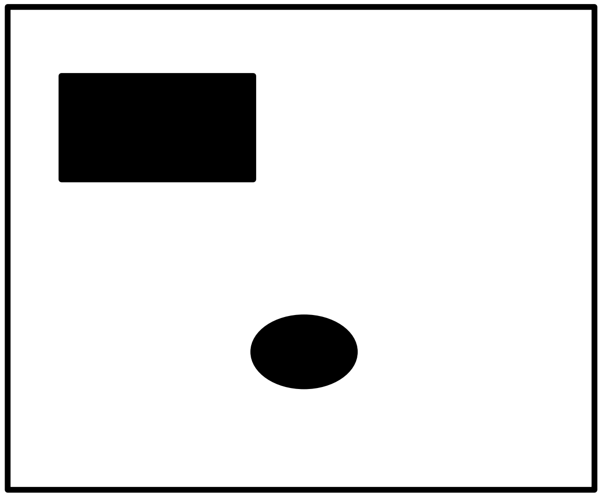
Get The Code

- If you have not already performed the recursive floodfill exercise on Vocareum or your own machine, please get the code:
- Vocareum Assignment: Sandbox – Recursion
- Download code to your own machine
  - Create a folder and at the terminal 'cd' to that folder
  - `wget http://ee.usc.edu/~redekopp/cs103/floodfill.tar`
  - `tar xvf floodfill.tar`

Flood Fill

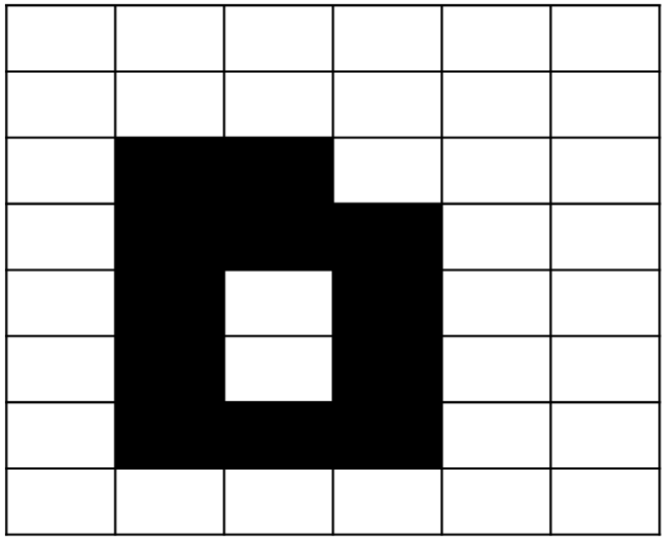
- Imagine you are given an image with outlines of shapes (boxes and circles) and you had to write a program to shade (make black) the inside of one of the shapes. How would you do it?
- Flood fill is a recursive approach
- Given a pixel
  - Base case: If it is black already, stop!
  - Recursive case: Call floodfill on each neighbor pixel
  - Hidden base case: If pixel out of bounds, stop!





Recursive Flood Fill

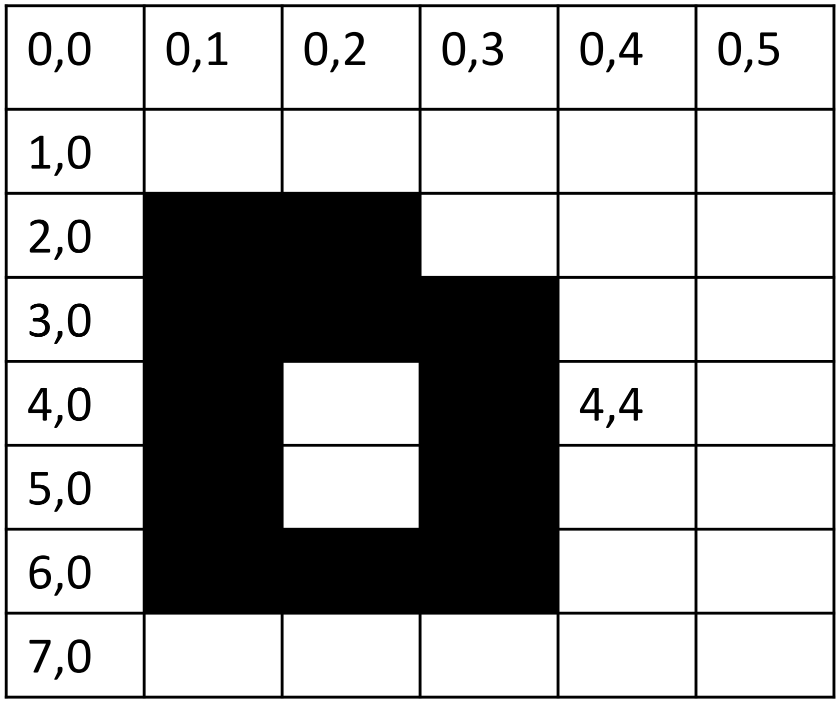
- Recall the recursive algorithm for flood fill?
  - Base case: black pixel, out-of-bounds
  - Recursive case: Mark current pixel black and then recurse on your neighbors



```
void flood_fill(int r, int c)
{
  if(r < 0 || r > 255 )
    return;
  else if ( c < 0 || c > 255) {
    return;
  }
  else if(image[r][c] == 0) {
    return;
  }
  else {
    // set to black
    image[r][c] = 0;
    flood_fill(r-1,c);  // north
    flood_fill(r,c-1);  // west
    flood_fill(r+1,c);  // south
    flood_fill(r,c+1);  // east

  }
}
```

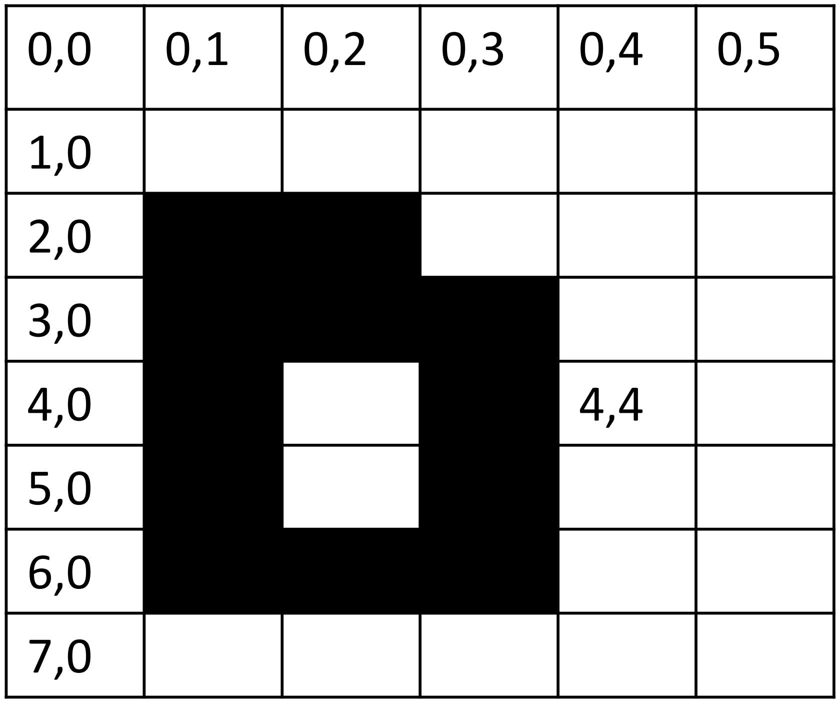
Recursive Ordering

- Give the recursive ordering of all calls for recursive flood fill assuming N, W, S, E exploration order starting at 4,4
  - From what square will you first explore to the west?
  - From what square will you first explore south?
  - From what square will you first explore east?
  - What is the maximum number of recursive calls that will be alive at any point in time?



Recursive Ordering

- Give the recursive ordering of all calls for recursive flood fill assuming N, W, S, E exploration order starting at 4,4
  - From what square will you first explore to the west?
  - From what square will you first explore south?
  - From what square will you first explore east?
  - What is the maximum number of recursive calls that will be alive at any point in time?
  - Notice recursive flood fill goes deep before it goes broad
  - Also notice that each call that is not a base case will make 4 other recursive calls



Developing Recursive Algorithms

Recursive Approach

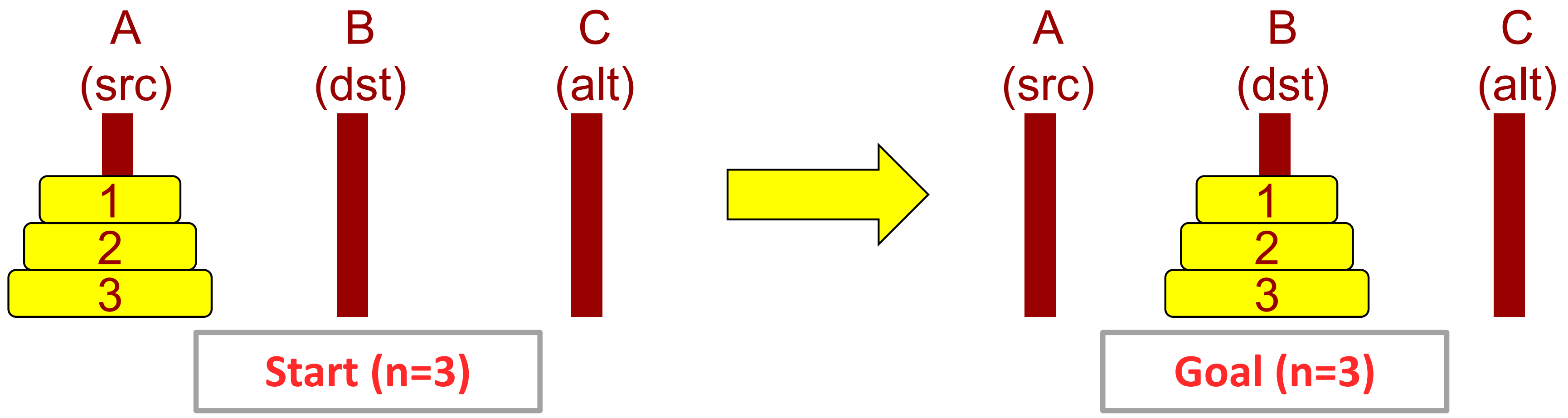
Steps to developing recursive algorithms & then coding them

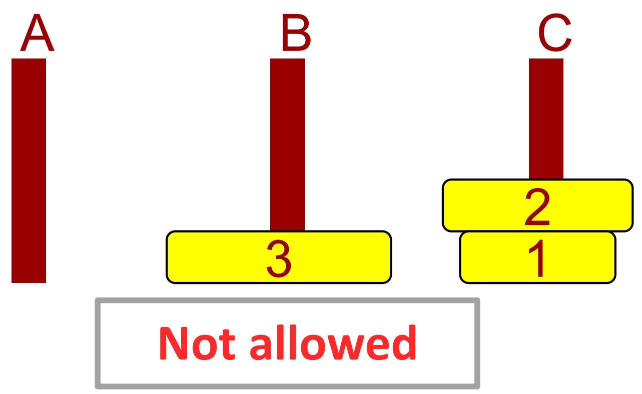
- Identify the recursive structure
  - How can a large version of the problem be solved with solutions to smaller versions of the problem
  - What do we need to do BEFORE recursing (i.e. what am I responsible for, what information do I need to extract, how to I create the smaller problem, etc.)?
  - What do we need to do AFTER we return from recursing (i.e. how do I take the smaller solution I get and combine it with the information I extracted to generate the bigger solution)?
- Identify base cases (i.e. when to stop)
- Ensure each recursive call makes progress toward one base case (i.e. avoid infinite recursions)

TOWERS OF HANOI

Towers of Hanoi Problem

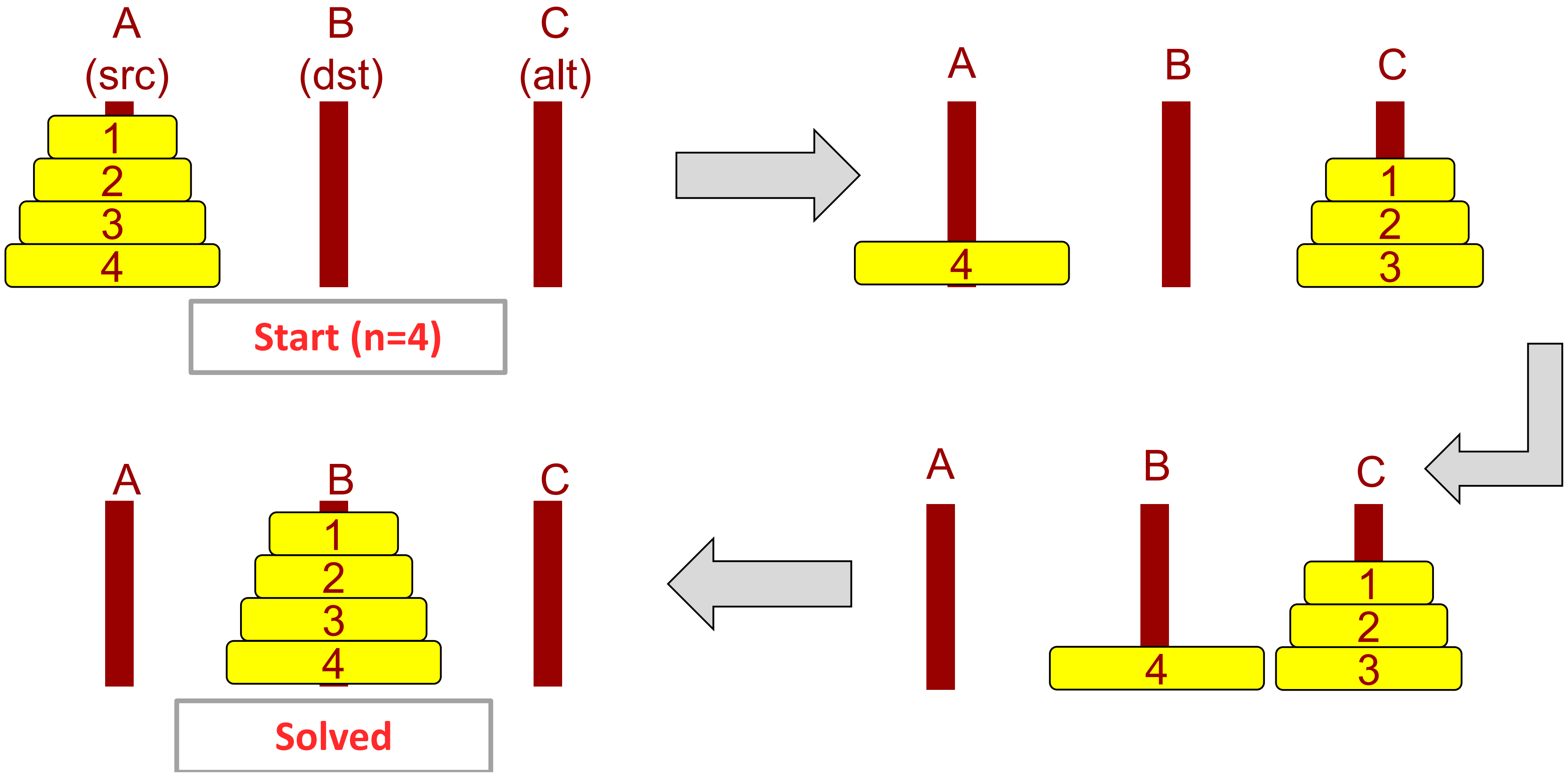
- Problem Statements: Move n discs from source pole to destination pole (with help of a 3rd alternate pole)
  - Can only move one disc at a time
  - **CANNOT** place a LARGER disc on top of a SMALLER disc





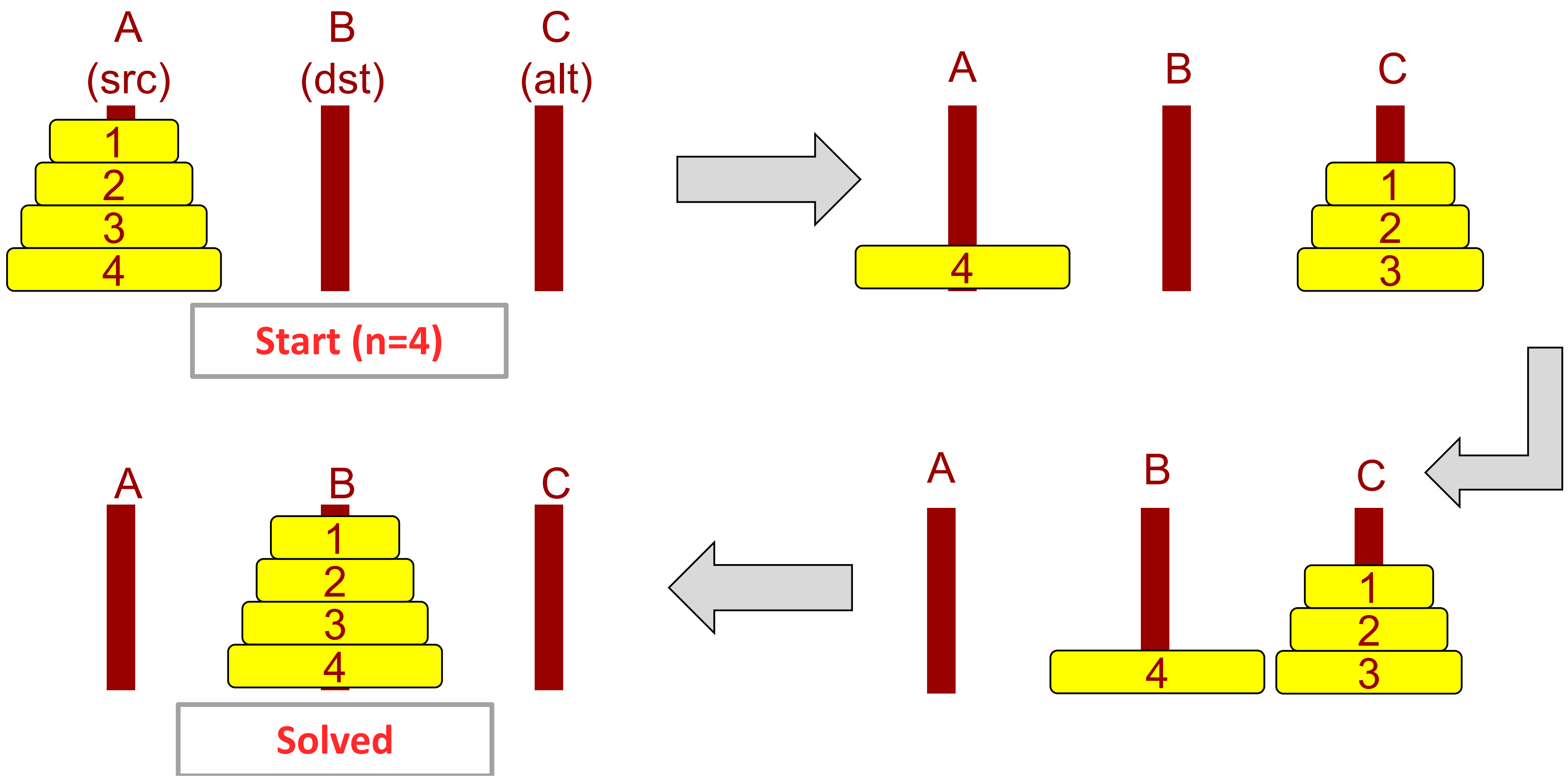
Finding Recursive Structure (1)

- Moving **n** discs to the destination starts with the task of moving **n-1** discs to the alternate



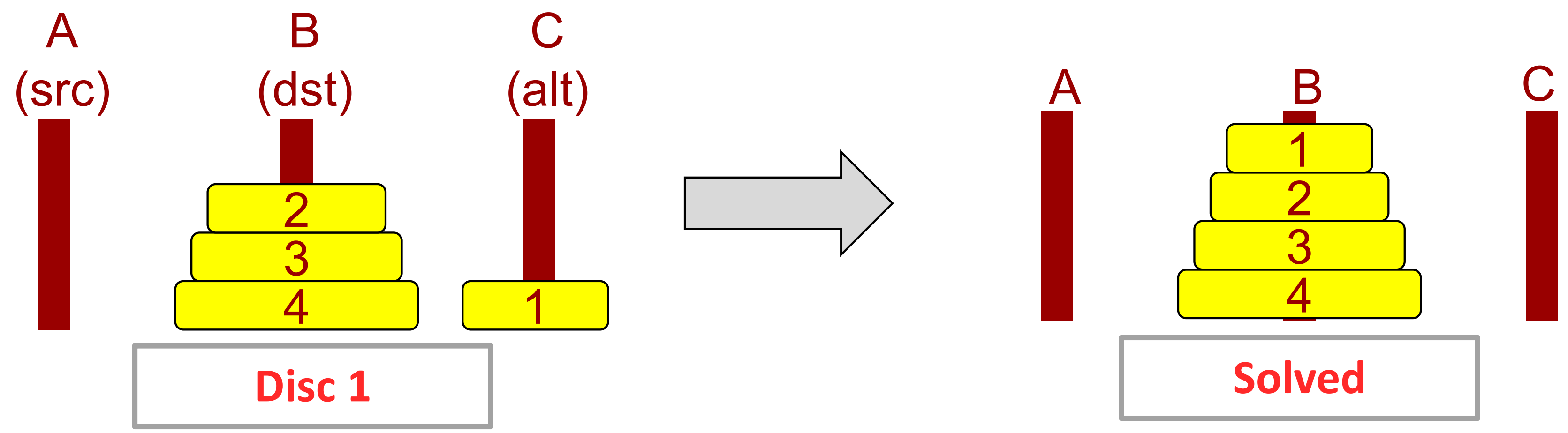
Defining Recursive Case

```
Recursive case:
1. Move n-1 discs from SRC to ALT <-- recursive call
2. Move disc n from SRC to DST    <-- work on disc you are responsible for
3. Move n-1 discs from ALT to SRC <-- recursive call
```



Defining Base Case

```
Base case:
1. Smallest disc (n=1) can always be moved from SRC to DST
```



Finding Recursive Function Signature

- What changes per call
  - Number of discs to move
  - Pole locations: SRC, DST, ALT
- Signature
  - `void towers(int n, char src, char dst, char alt);`
- Base case: when n is 1
  - `Print "Move disc 1 from` *`src`* `to` *`dst`*`"`
- Recursive case
  - `Recurse:  towers(n-1, src, alt, dst);`
  - `Print "Move disc n from` *`src`* `to` *`dst`*`"`
  - `Recurse:  towers(n-1, alt, dst, src);`

Exercise

- Implement the Towers of Hanoi code
  - Vocareum: Recursion-2
  - Or on your VM
    - $ wget http://ee.usc.edu/~redekopp/cs103/hanoi.cpp
  - Just print out "`move disc=x from y to z`" rather than trying to "move" data values
    - Move disc 1 from a to b
    - Move disc 2 from a to c
    - Move disc 1 from b to c
    - Move disc 3 from a to b
    - Move disc 1 from c to a
    - Move disc 2 from c to b
    - Move disc 1 from a to b

Recursive Box Diagram

Towers Function Prototype

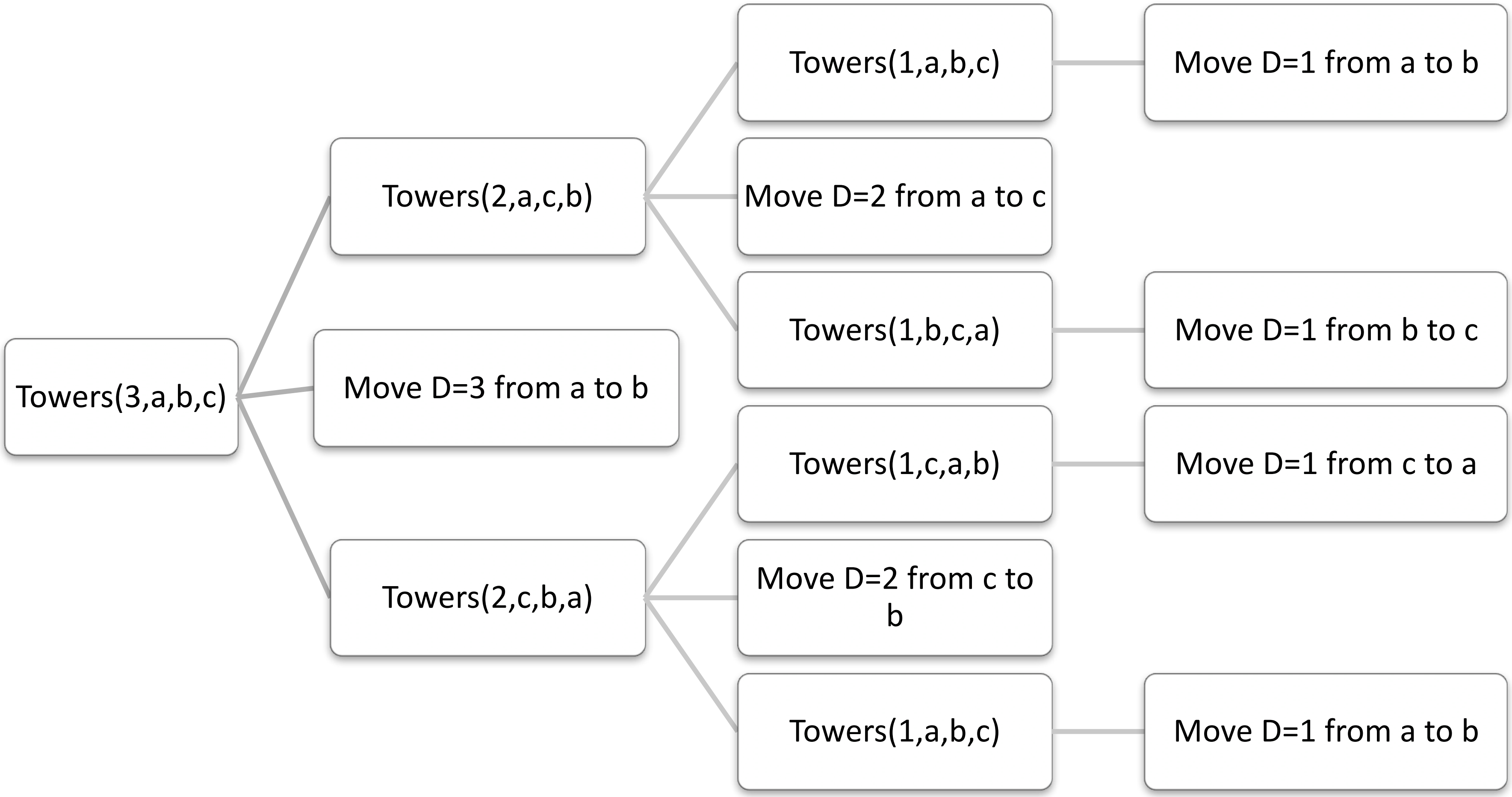
```
towers(disc,src,dst,alt)
```

Base case: when n is 1

```
    Print "Move disc 1 from src to dst"
```

Recursive case

```
    Recurse:  towers(n-1, src, alt, dst);
    Print "Move D=<n> from <src> to <dst>"
    Recurse:  towers(n-1, alt, dst, src);
```



Convert a single integer to a queue (deque) of individual integer digits

INT TO DIGITS

Problem Statement and Approach

- Write a recursive function to convert a single positive integer into a deque of the individual integer digits.

Input 12658

Desired result

0	1	2	3	4
1	2	6	5	8

Approach

Step 1 1265

result

0
8

Step 2 126

result

0	1
5	8

…

…

Finding Recursive Solutions

- **Identify the recursive structure**
  - **How can a large version of the problem be solved with solutions to smaller versions of the problem?**
  - What **1 thing** is each recursive call responsible for
  - What do we need to do **BEFORE** recursing?
  - What do we need to do **AFTER** we return from recursing?
- **Identify base cases (i.e. when to stop)**
- Ensure each recursive call makes progress toward one base case

Deriving a Solution

- Identify the base case
  - What trivial version of the problem can be easily solved?
- Recursive case:
  - What 1 thing is each recursion responsible for?
  - How do I extract one digit? Which digit?
  - Where do I put that digit? Front or back of result?
  - How do I make the problem smaller?

Input 12658

Desired result

0	1	2	3	4
1	2	6	5	8

Approach

Step 1 1265

result

0
8

Deriving a Solution

- Identify the base case
  - What trivial version of the problem can be easily solved? ______________________
- Recursive case:
  - What 1 thing is each recursion responsible for? ______________________
  - How do I extract one digit? Which digit? ______ __________________
  - Where do I put that digit? Front or back of result? _____________________
  - How do I make the problem smaller? _________________________

```
void digits(
  unsigned int n,
  deque<int>& res)
{










}
```

Input 12658

Desired result

0	1	2	3	4
1	2	6	5	8

Discussion (1)

- What if we recursed first and isolated the digit after returning from the recursion.
- Update the code using this approach

```
void digits(
  unsigned int n,
  deque<int>& res)
{









}
```

Input

12658

Desired result

0	1	2	3	4
1	2	6	5	8

Discussion (2)

- How would main() be written to use digits()
- Why did we pass by the result deque by reference?
  - Challenge: Recode the solution using the signature, `deque<int> digits(unsigned int n);` thinking carefully about where copies of the deque are made

```
void digits(unsigned int n,
            deque<int>& res);

int main()
{
  int x;   cin >> x;


  // call digits


}
```

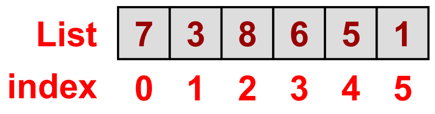
```
deque<int> digits(
    unsigned int n)
{
  if(n < 10) {
    deque<int> x;
    x.push_front(n);
    return x;
  }
  else {
    deque<int> x =
      digits(n);
    x.push_back(n%10);
    return x;
  }

}
```

Recursive Bubblesort and Mergesort

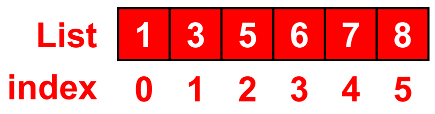
SORTING

Sorting

- How can sorting be formulated recursively?
  - Actually many ways! Can you think of an easy way?
- Many sorting algorithms of differing complexity (i.e. faster or slower)
- Bubble Sort – $O(n^2)$ runtime
  - On each pass through thru the list, move the maximum element to the end of the list.
  - Then __________ using a list of size ______



Original

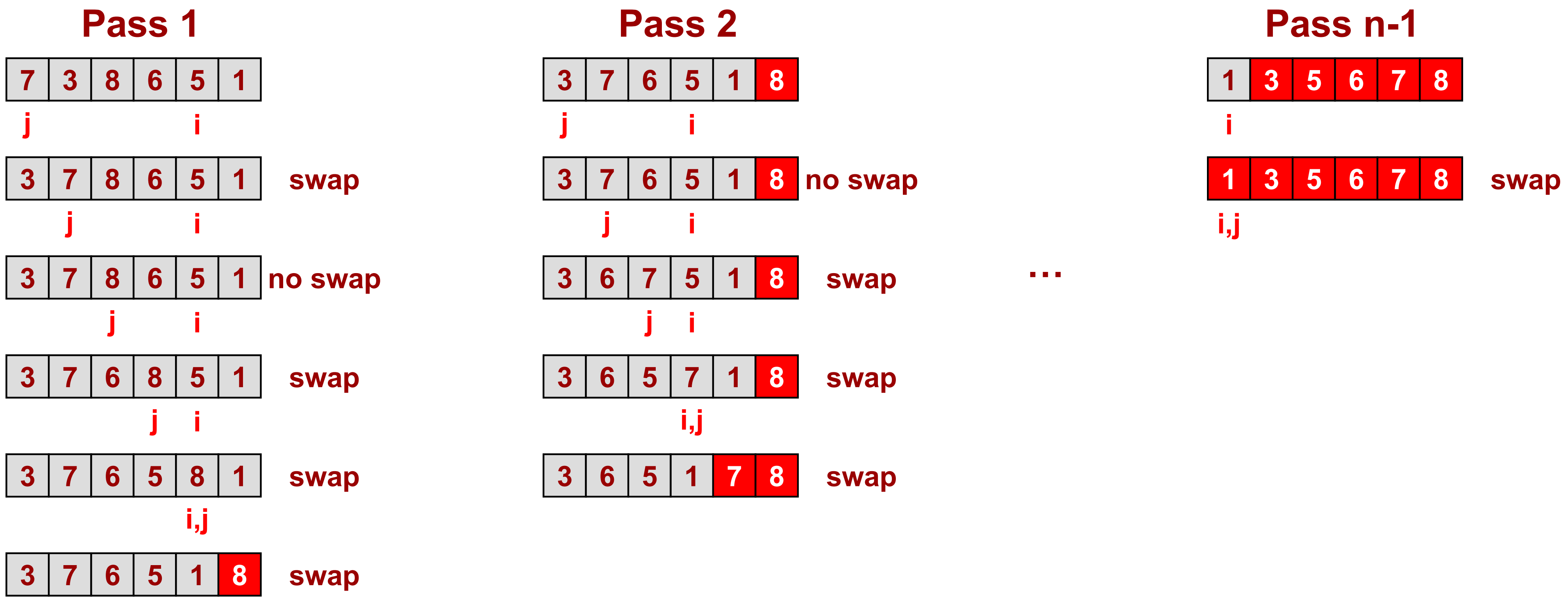


Final answer

Iterative Bubble Sort Algorithm

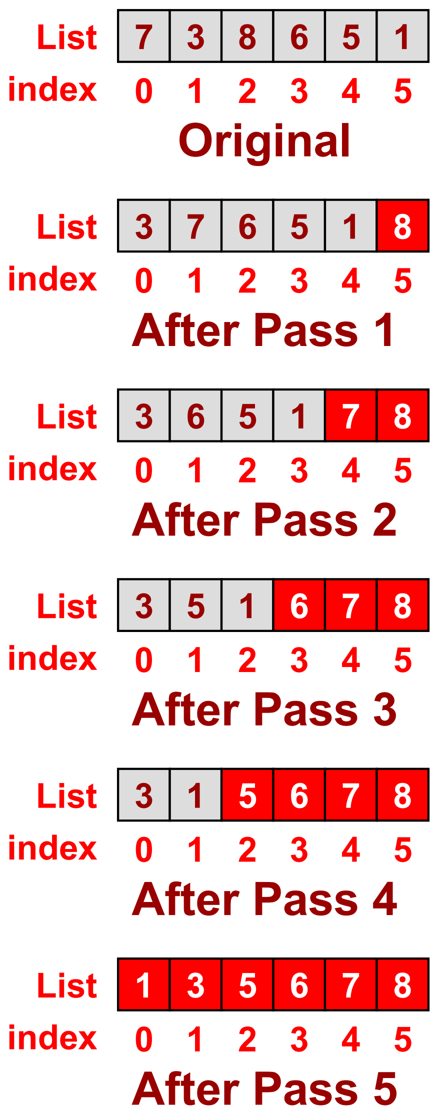
```
n ← length(List);
for( i=n-2; i >= 1; i--)
  for( j=1; j <= i; j++)
    if ( List[j] > List[j+1] ) then
      swap List[j] and List[j+1]
```

Bubblesort requires O(n^2) time!



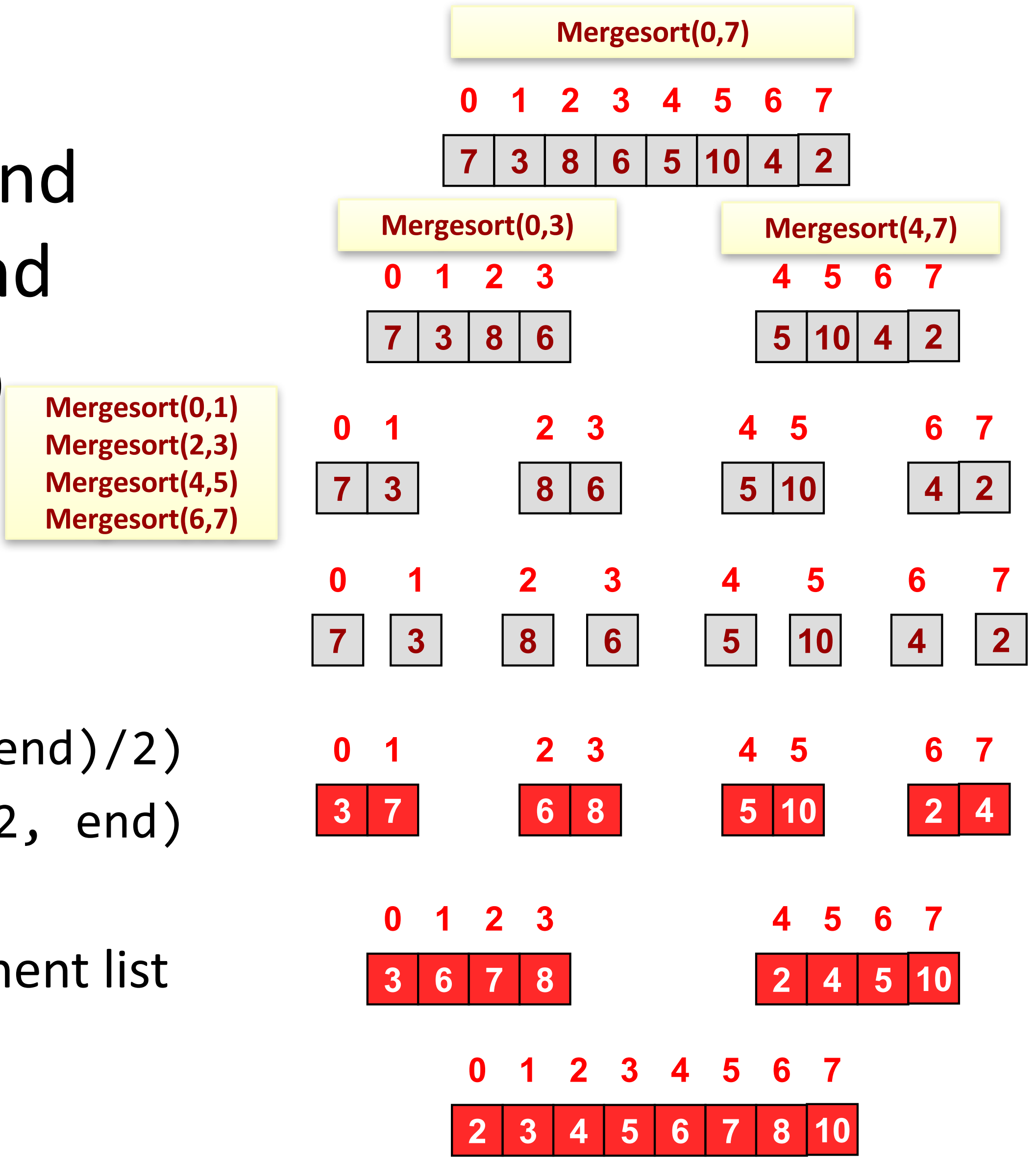
Sorting

- Bubble Sort – O(n^2) runtime
  - On each pass through thru the list, move the maximum element to the end of the list.
  - Then repeat/recurse on a list of size (n-1)



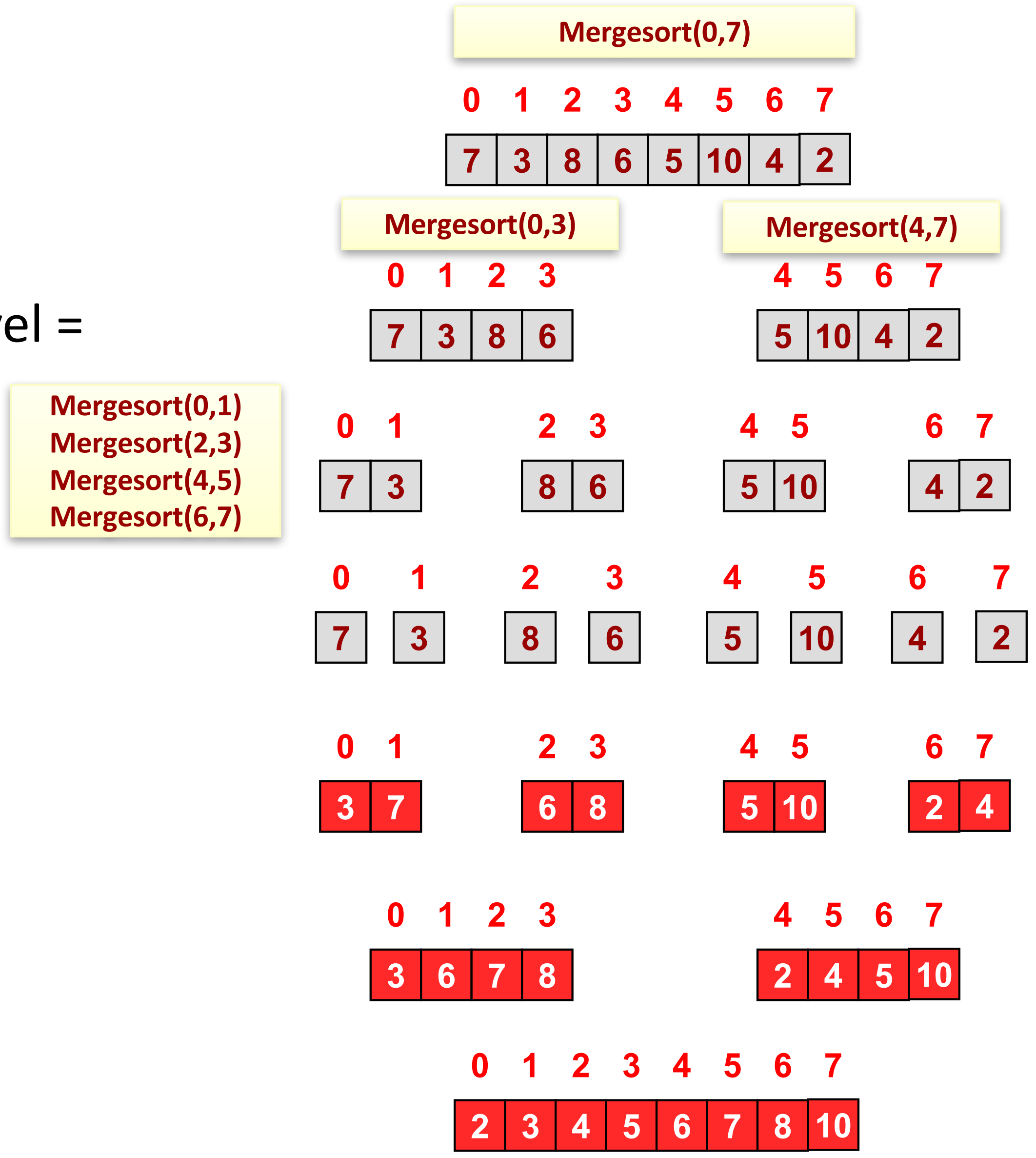
Recursive Sort (MergeSort)

- Break sorting problem into smaller sorting problems and merge the results at the end
- `mergesort(start,end)`
  - if remaining list is size 1
    - return
  - else
    - `mergesort(start, (start+end)/2)`
    - `mergesort(1+(start+end)/2, end)`
    - Merge each sorted list of n/2 elements into a sorted n-element list



Recursive Sort (MergeSort)

- Run-time analysis
  - # of recursion levels =
    - ____________
  - Total operations to merge each level =
    - ___ operations total to merge two lists over all recursive calls
- mergesort = O(_______________)

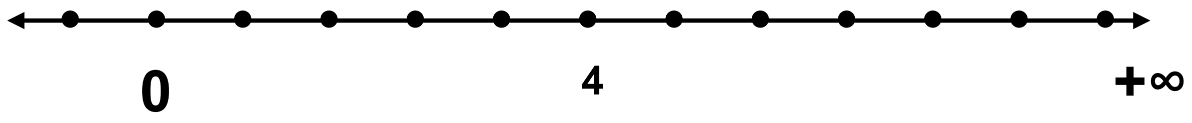


FINDING THE SQUARE ROOT

Square Root Finder

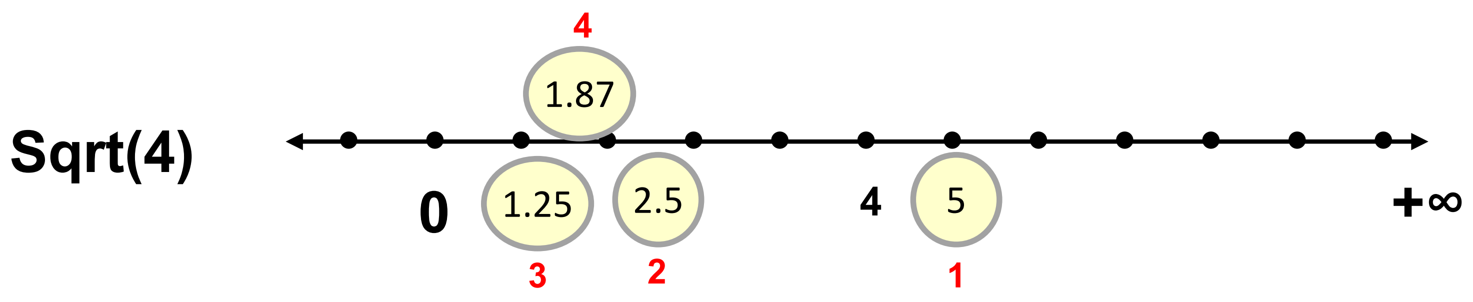
- Suppose we did not have the `sqrt(x)` function available in the math library
- How could we develop an algorithm to find the square root
- What is the range of possible answers to `sqrt(x)`?
  - Is the square root of x always smaller than x?
  - We can certainly bound `sqrt(x)` by ______________
- How can we find the `sqrt(x)`?
  - We could just start guessing by picking values, n, and squaring them to see if they are close to `x (within some ε)`

Sqrt(4)



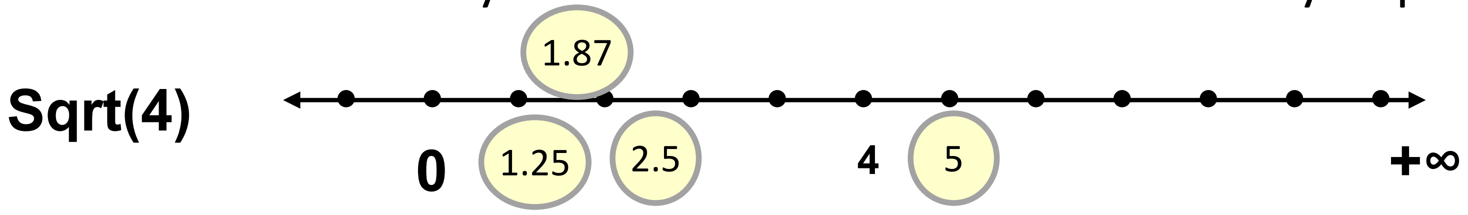
Square Root Finder

- Suppose we did not have the `sqrt(x)` function available in the math library
- How could we develop an algorithm to find the square root
- What is the range of possible answers to `sqrt(x)`?
  - Is the square root of x always smaller than x?
  - We can certainly bound `sqrt(x)` by [0, x+1]
- How can we find the `sqrt(x)`?
  - We could just start guessing by picking values, n, and squaring them to see if they are close to x `(within some ε)`
  - To be more efficient we could use a binary search of the number line



Recursive Helper Functions

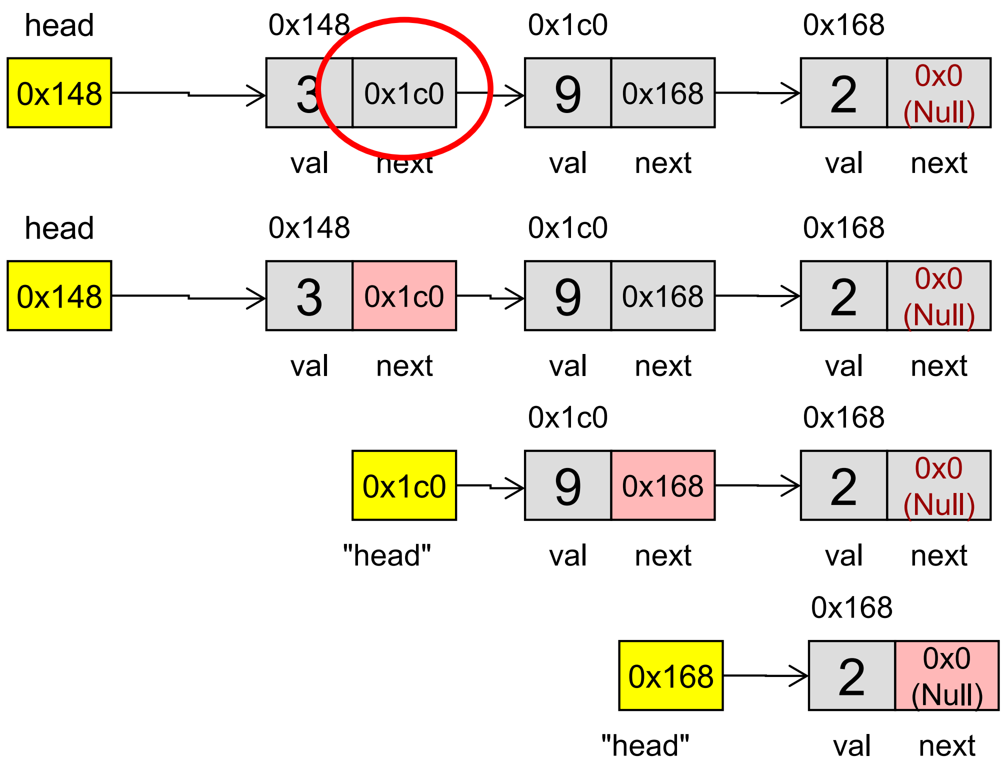
- Sometimes we want to provide a user with a simple interface (arguments, etc.), but to implement it recursively we need additional arguments to our function
- In that case, we often let the top-level, simple function call a recursive "**helper**" function that provides the additional arguments needed to do the work
  - `double sqrt(double x);` // User Interface
  - `double sqrt(double x, double lo, double hi);` // Helper
- In-class-exercise: **sqrt**
  - Find the square root of, x, without using sqrt function…
  - Pick a number, square it and see if it is equal to x
  - Use a binary search to narrow down the value you pick to square



RECURSION & LINKED LISTS

Linked Lists and Recursion

- Consider a linked list with a head pointer
- If you were given the pointer at `head->next`, isn't that a "head" pointer to the n-1 other items in the list?



Exercises

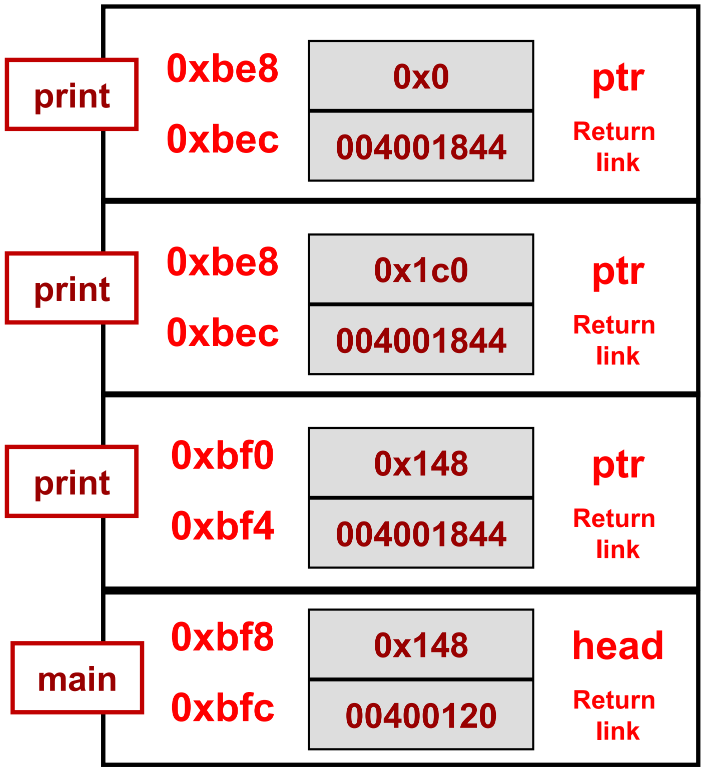
- In-Class exercises
  - Monkey_recurse
  - Monkey_recback
  - List_max
  - Monkey_reverse

Childs toy "Barrel of Monkeys" let's children build a chain of monkeys that can be linked arm in arm

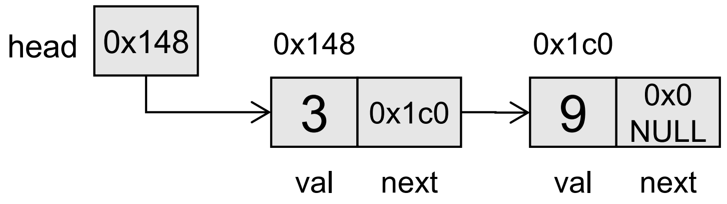
http://www.toysrus.com/graphics/tru_prod_images/Barrel-of-Monkeys-Game----pTRU1-2907042dt.jpg

Recursive Operations on Linked List

- Many linked list operations can be recursively defined
- Can we make a recursive iteration function to print items?
  - Recursive case: Print one item then the problem becomes to print the n-1 other items.
    - Notice that any 'next' pointer can be though of as a 'head' pointer to the remaining sublist
  - Base case: Empty list (i.e. Null pointer)



```
void print(Item* ptr)
{
  if(ptr == NULL) return;
  else {
    cout << ptr->val << endl;
    print(ptr->next);
  }
}
int main()
{ Item* head;
  ...
  print(head);
}
```



Generating All Combinations Using Recursion

Making multiple recursive calls

Recursion's Power

- The power of recursion often comes when each function instance makes ***multiple*** recursive calls
- As you will see this often leads to exponential number of "combinations" being generated/explored in an easy fashion

```
void rfunc1(int n)
{
  ...
  rfunc1(n-1);
  ...
}
```

1 Recursive Call

```
void rfunc2(int n)
{
  ...
  t = rfunc2(n-1);
  s = rfunc2(n-2);
  ...
}
```

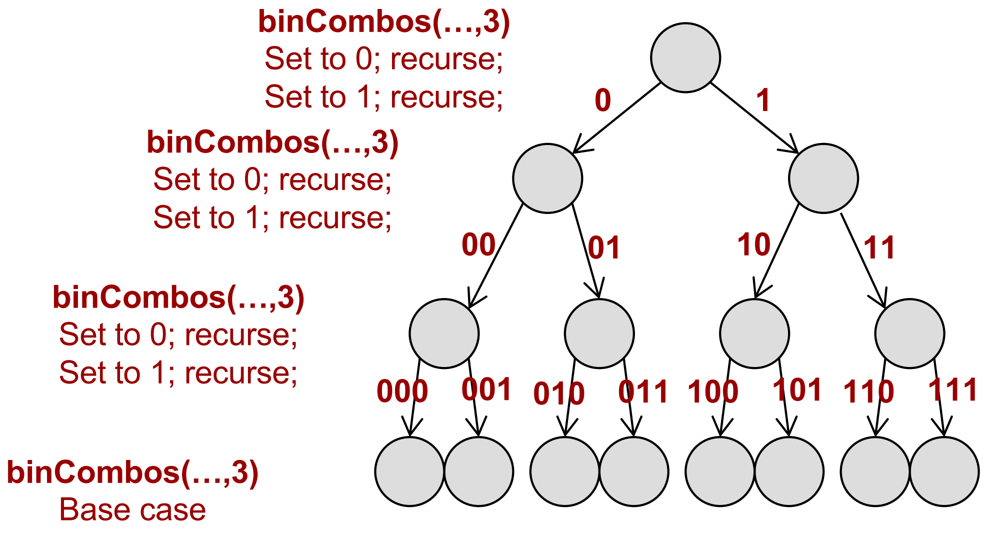
Multiple Recursive Calls

Binary Combinations

- If you are given the value, n, and a string with n characters could you generate all the combinations of n-bit binary?
- Do so recursively!

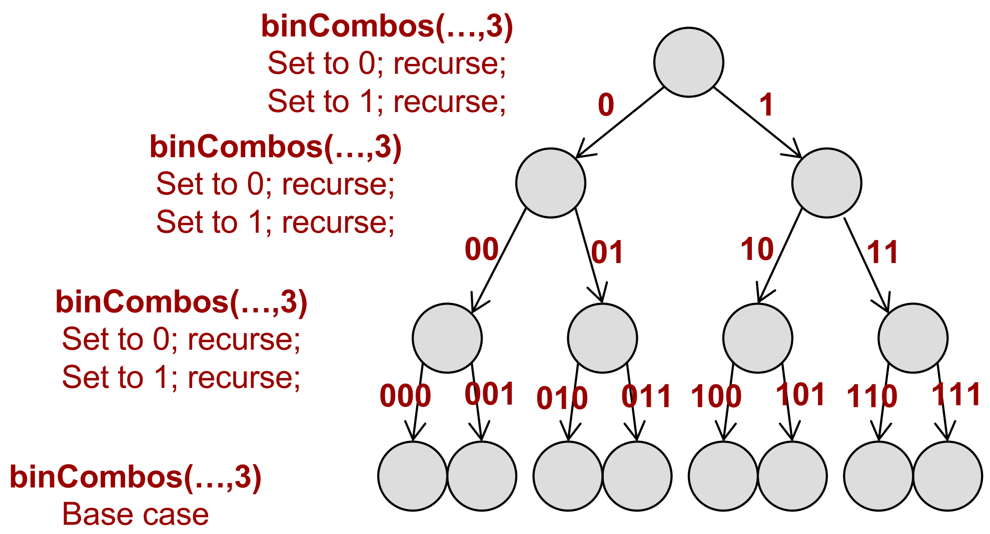
Exercise: bin_combo_str

1-bit Bin.	2-bit Bin.	3-bit Bin.	4-bit Bin.
0	00	000	0000
1	01	001	0001
	10	010	0010
	11	011	0011
		100	0100
		101	0101
		110	0110
		111	0111
			1000
			1001
			1010
			1011
			1100
			1101
			1110
			1111



Recursion and DFS

- Recursion forms a kind of Depth-First Search



```
// user interface
void binCombos(int len)
{
  binCombos(_____________);
}
// helper-function
void binCombos(string prefix,
               int len)
{









}
```

In-Class Activity

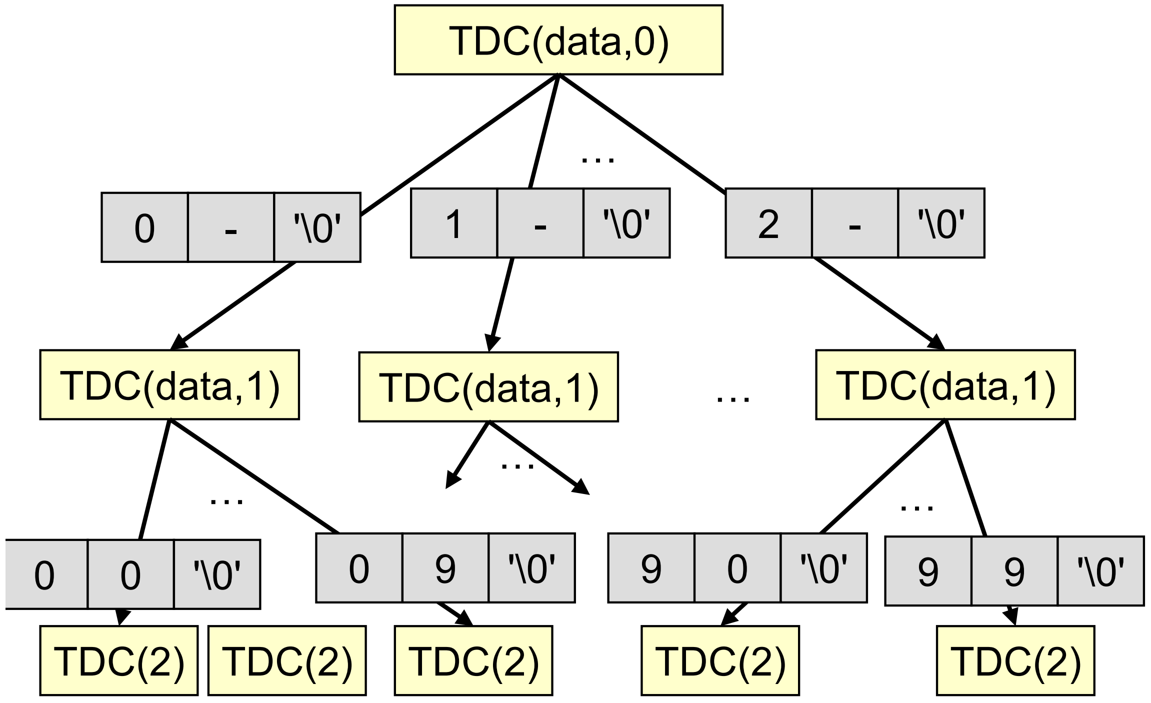
- Generate all 3-length string combinations using letters: A, B

- Form groups of 3 people (choose who is person 1, 2, and 3)
- Draw the array below on one piece of paper for the whole group
- If you're array entry is:
  - **Blank**, write 'A' and pass to person i+1 if you can
  - **Has 'A'**, erase and change to 'B' and pass to person i+1 if you can
  - **Has 'B'**, erase and pass back to person i-1 if you can

1	**2**	**3**

Generating All Combinations

- Recursion offers a simple way to generate all combinations of N items from a set of options, S
  - Example: Generate all 2-digit decimal numbers (N=2, S={0,1,…,9})



```
void TwoDigCombos(string data,
                  int curr)
{
  if(curr == 2 )
    cout << data;
  else {
    for(int i=0; i < 10; i++){
      // set to i
      data += '0' + (char)i;
      // recurse
      TwoDigCombos(data, curr+1);
    }
  }
}
```

Recursion and Combinations

- Consider the problem of generating all 2-length combinations of a set of values, S.
  - Ex. Generate all length-n combinations of the letters in the set S={'U','S','C'} (i.e. UU, US, UC, SU, SS, SC, CU, CS, CC)
  - How could you do it with loops (how many loops would you need)?
- Consider the problem of generating all 3-length combinations of a set of values, S.
  - Ex. Generate all length-n combinations of the letters in the set S={'U','S','C'} (i.e. UUU, UUS, UUC, USU, USS, USC, etc.)
  - How many loops would you need?
- Consider the problem of generating all n-length combinations of a set of values, S.
  - How many loops would you need? Is that even possible?

```
void usccombos2()
{
  char str[3] = "--";
  char vals[3] = {'U','S','C'};
  for(int i=0; i != 3; i++){
    str[0] = vals[i];
    for(int j=0; j != 3; j++){
      str[1] = vals[j];
      cout << str << endl;
    }
  }
}
```

0	1	
-	-	\0

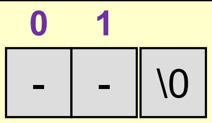
```
void usccombos3()
{
  char str[4] = "---";
  char vals[3] = {'U','S','C'};
  for(int i=0; i != 3; i++){
    str[0] = vals[i];
    for(int j=0; j != 3; j++){
      str[1] = vals[j];
      for(int k=0; k != 3; k++){
        str[2] = vals[k];
  }
}
```

0	1	2	
-	-	-	\0

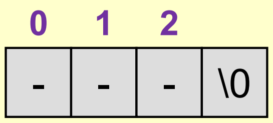
Recursion and Combinations

- Consider the problem of generating all n-length combinations of a set of values, S.
  - How many **loops** would you need?
  - Is that even possible?

```
void usccombos2()
{
  char str[3] = "--";
  char vals[3] = {'U','S','C'};
  for(int i=0; i != 3; i++){
    str[0] = vals[i];
    for(int j=0; j != 3; j++){
       str[1] = vals[j];
       cout << str << endl;
    }
  }
}
```



```
void usccombos3()
{
  char str[4] = "---";
  char vals[3] = {'U','S','C'};
  for(int i=0; i != 3; i++){
    str[0] = vals[i];
    for(int j=0; j != 3; j++){
       str[1] = vals[j];
       for(int k=0; k != 3; k++){
          str[2] = vals[k];
  }
}
```



Finding the Recursive Structure

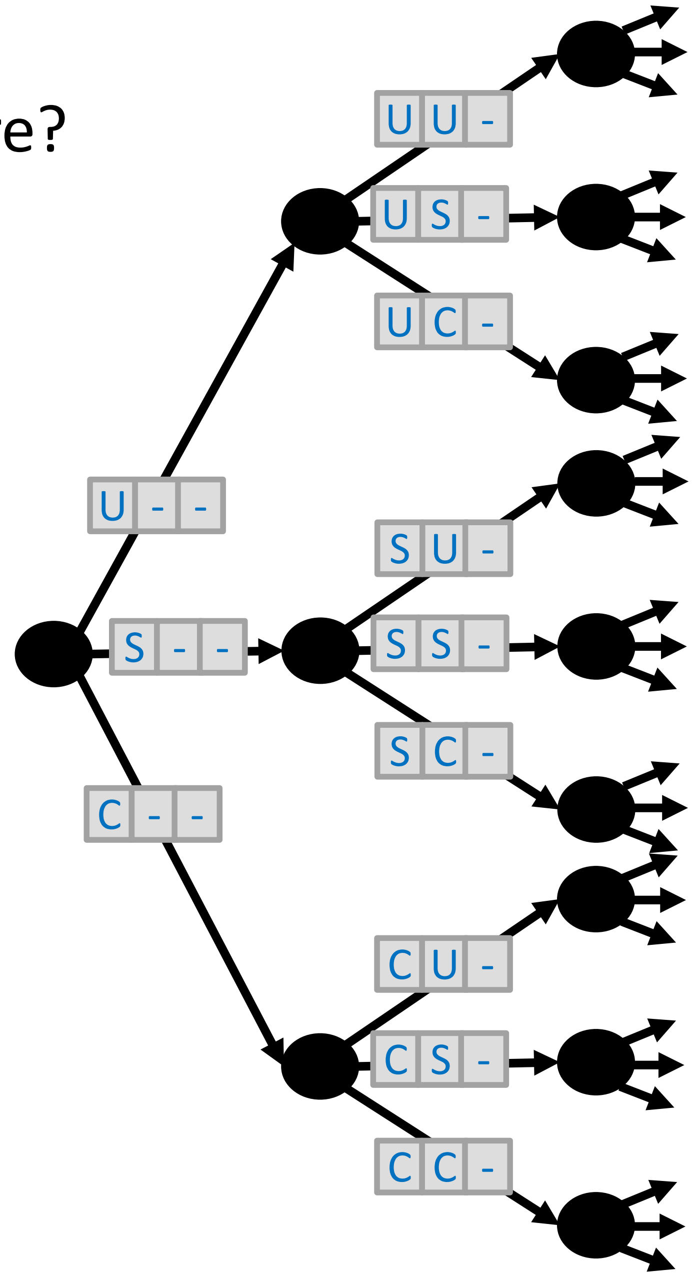
- Where do you see the recursive structure?

U
S
C

n=1

n=2

n=3

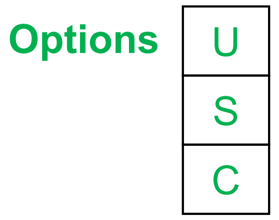


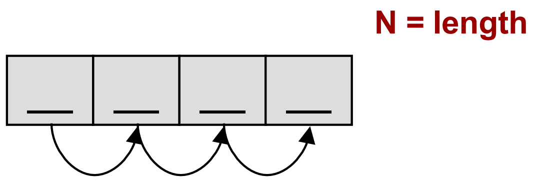
Recursion and Combinations

- Recursion provides an elegant way of generating all **n**-length combinations of a set of values, S.
  - Ex. Generate all length-**n** combinations of the letters in the set **S**={'U','S','C'}
  - You would need **n** loops. But we don't have a way of executing a "variable" number of loops…Oh wait! We can use recursion!
- General approach:
  - Need some kind of **array/vector/string** to store partial answer as it is being built
  - Each recursive call is only responsible for one of the **n** "places" (say location, **i**)
  - The function will iteratively (loop) try each option in **S** by setting location **i** to the current option, then recurse to handle all remaining locations (i+1 to n)
    - Remember you are responsible for only one location
  - Upon return, try another option value and recurse again
  - Base case can stop when all n locations are set (i.e. recurse off the end)
  - Recursive case returns after trying all options

Coding a Solution

- Generate all string combinations of length n from a given list (vector) of characters





Use recursion to walk down the 'places'
At each 'place' iterate through & try all options

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void all_combos(vector<char>& letters,
                int n)
{

}

int main() {
   vector<char> letters;
   letters.push_back('U');
   letters.push_back('S');
   letters.push_back('C');

   all_combos(letters, 2);

   all_combos(letters, 4);

   return 0;
}
```

Exercises

- bin_combos_str
- basen_combos
- all_letter_combos
- zero_sum

Knapsack Problem

- Knapsack problem
  - You are a traveling salesperson. You have a set of objects with given weights and values. Suppose you have a knapsack that can hold N pounds, which subset of objects can you pack that maximizes the value.
  - Example:
    - Knapsack can hold 35 pounds
    - Object A: 7 pounds, $12.50 ea. Object B: 10 pounds, $18 ea.
    - Object C: 4 pounds, $7 ea. Object D: 2.4 pounds, $4 ea.
- Let's solve a simpler version of generating all the combinations of objects that would fit in a given weight (don't worry about duplicates due to order…A,A,B vs. B,A,A or A,B,A)
- Get the code:
  - Vocareum: Sandbox - Recursion 2
  - VM/Laptop: $ wget http://ee.usc.edu/~redekopp/cs103/knapsack.cpp

Ignore unless told otherwise.

BACKUP

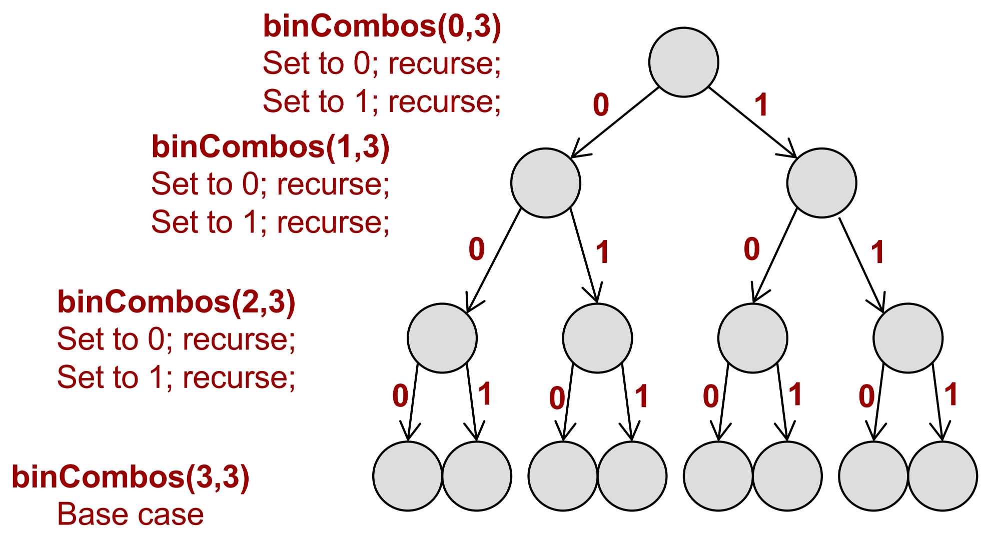
Recursion Analysis

- What would this code print for
  - X=3, y=2
  - X=10, y=1
  - X=2, y=3

```
#include <iostream>
#include <string>
using namespace std;

void mystery(int r, string pre, int n) {
   if(pre.length() == n){
      cout << pre << endl;
   }
   else {
      for(int i=0; i < r; i++){
         char c = static_cast<char>('0'+i);
         mystery(r, pre + c, n);
      }
   }
}

int main() {
   int x, y;
   cin >> x >> y;

   string pre;

   mystery(x, pre, y);

   return 0;
}
```

Recursion and DFS (w/ C-Strings)

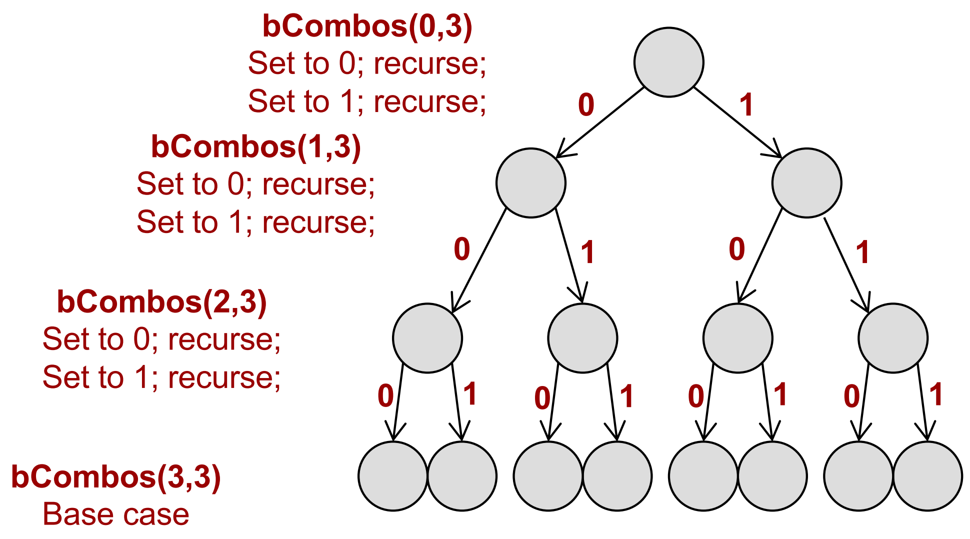
- Recursion forms a kind of Depth-First Search



```
void binCombos(char* data,
               int curr,
               int len)
{
  if(curr == len )
    data[curr] = '\0';
  else {
    // set to 0
    data[curr] = '0';
    // recurse
    binCombos(data, curr+1, len);
    // set to 1
    data[curr] = '1';
    // recurse
    binCombos(data, curr+1, len);
  }
}
```

Recursion and DFS (w/ C-Strings)

- Answer: All combinations of base x with y digits

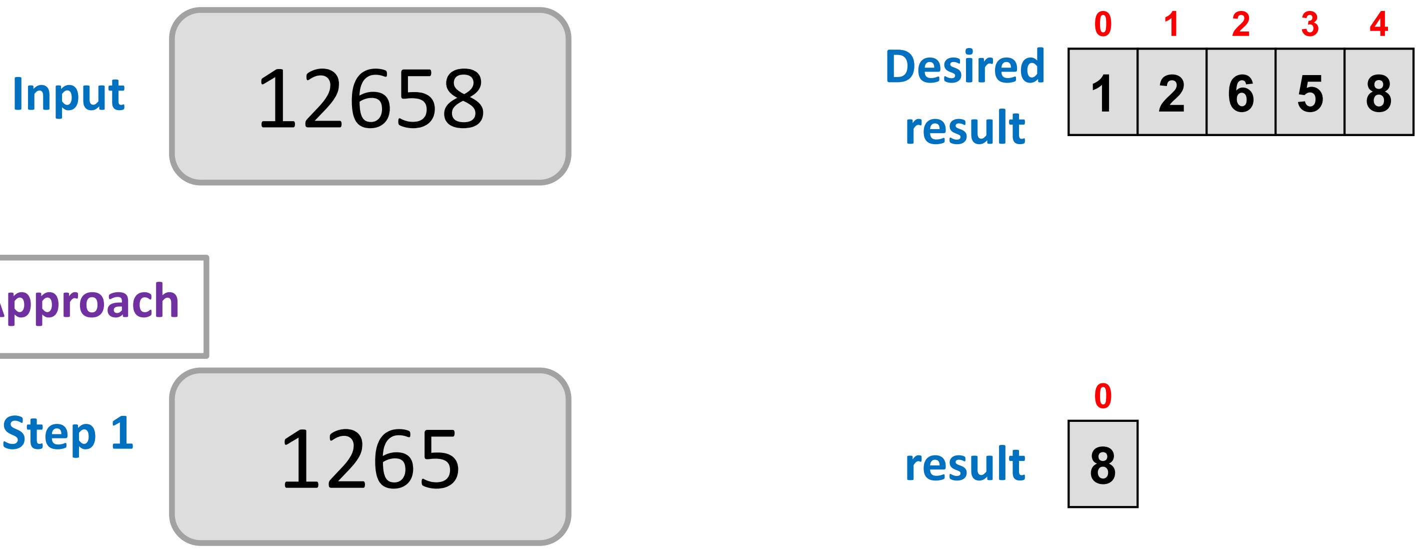


```
#include <iostream>
#include <string>
using namespace std;

void basen_combos(int r, string pre, int n) {
   if(prefix.length() == n){
      cout << pre << endl;
   }
   else {
      for(int i=0; i < r; i++){
         char c = static_cast<char>('0'+i);
         basen_combos(r, prefix + c, n);
      }
   }
}

int main() {
   int base, numDigits;
   cin >> x >> y;

   string pre;

   basen_combos(x, pre, y);

   return 0;
}
```

SOLUTIONS

Deriving a Solution

- Identify the base case
  - What trivial version of the problem can be easily solved? *1 digit num.*
- Recursive case:
  - What 1 thing is each recursion responsible for? *1 digit of the number*
  - How do I extract one digit? Which digit? *Easiest to find 1's digit using mod operator*
  - Where do I put that digit? Front or back of result? *Front of deque*
  - How do I make the problem smaller? *Divide by 10*



Deriving a Solution

- Identify the base case
  - What trivial version of the problem can be easily solved? *1 digit num.*
- Recursive case:
  - What 1 thing is each recursion responsible for? *1 digit of the number*
  - How do I extract one digit? Which digit? *Easiest to find 1's digit using mod operator*
  - Where do I put that digit? Front or back of result? *Front of deque*
  - How do I make the problem smaller? *Divide by 10*

```
void digits(
  unsigned int n,
  deque<int>& res)
{
  if( n < 10 ){
    res.push_front(n);
  }
  else {
    int d = n % 10;
    res.push_front(d);
    digits(n/10, res);
  }
}
```

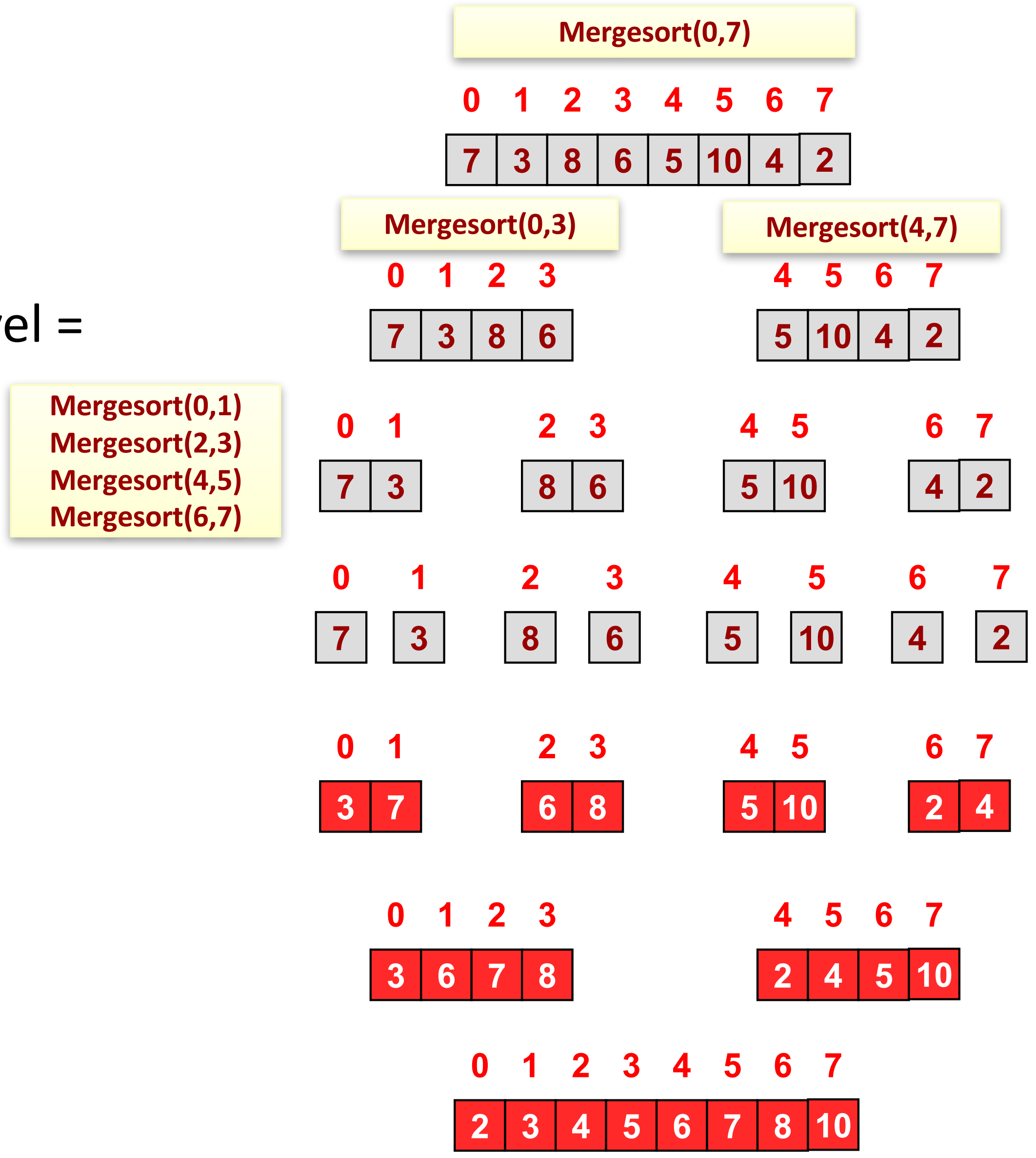
Input 12658

Desired result

0	1	2	3	4
1	2	6	5	8

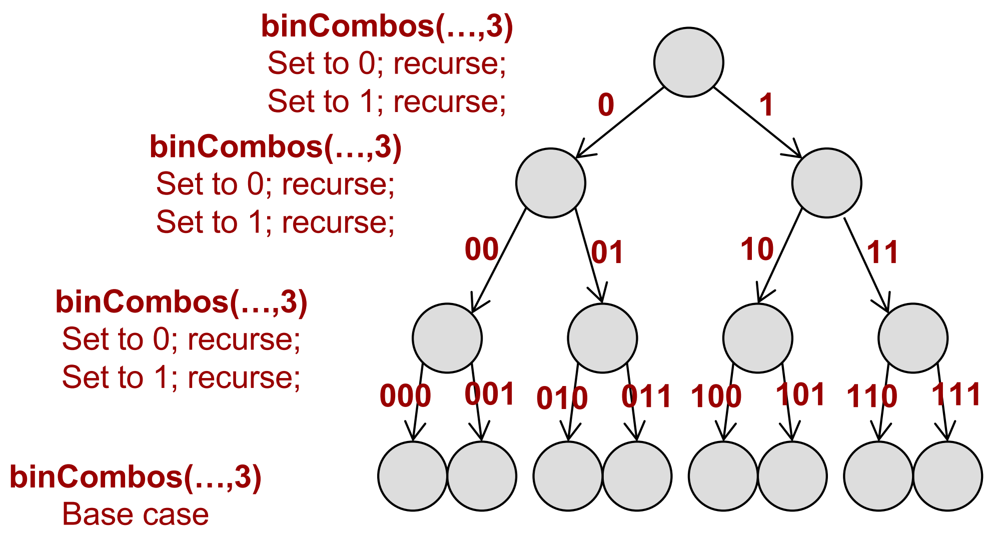
Recursive Sort (MergeSort)

- Run-time analysis
  - # of recursion levels =
    - $\log_2(n)$
  - Total operations to merge each level =
    - n operations total to merge two lists over all recursive calls
- mergesort = $O(n * \log_2(n))$



Recursion and DFS

- Recursion forms a kind of Depth-First Search



```
// user interface
void binCombos(int len)
{
  binCombos("", len);
}
// helper-function
void binCombos(string prefix,
               int len)
{
  if(prefix.length() == len )
    cout << prefix << endl;
  else {
    // recurse
    binCombos(data+"0", len);
    // recurse
    binCombos(data+"1", len);
  }
}
```

Finding the Recursive Structure

- Where do you see the recursive structure?

U
S
C

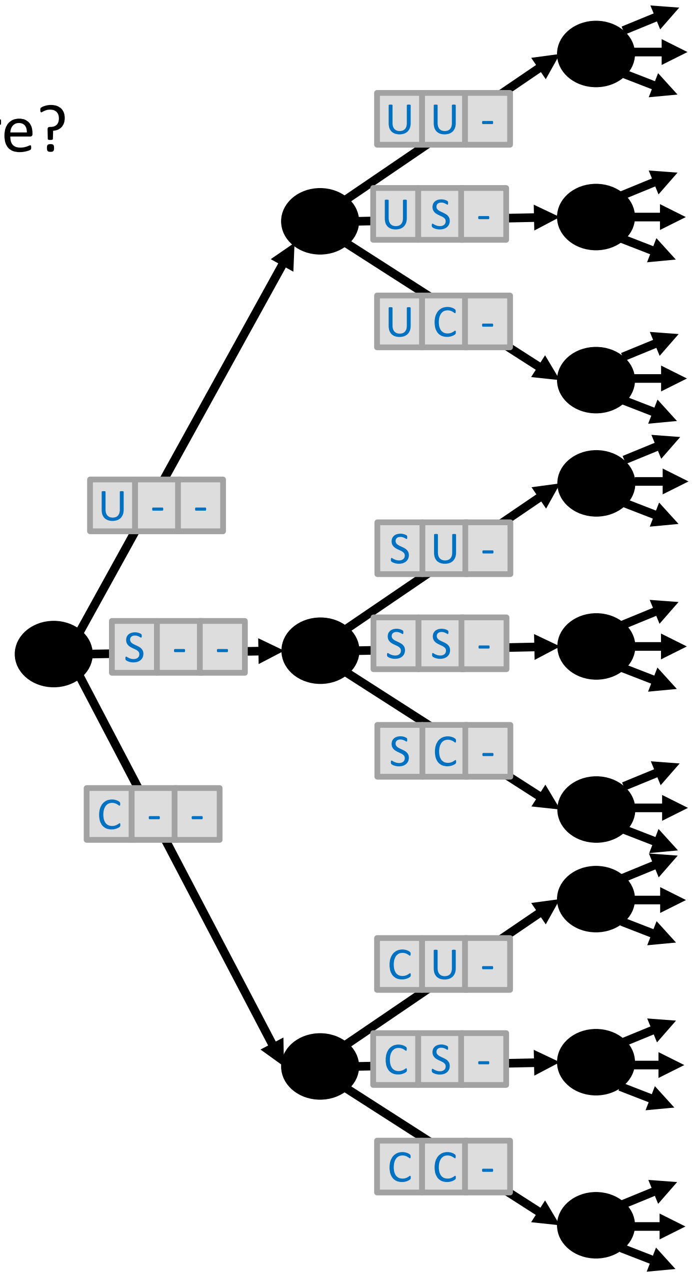
n=1

UU
US
UC
SU
SS
SC
CU
CS
CC

n=2

UUU
UUS
UUC
USU
USS
USC
UCU
UCS
UCC
SUU
SUS
. . .
CCC

n=3


U - -
S - -
C - -
U U -
U S -
U C -
S U -
S S -
S C -
C U -
C S -
C C -